Deleted: UDDI Spec TC

Schema Centric XML Canonicalization Version 1.0

Document identifier:

SchemaCentricCanonicalization

This version:

http://uddi.org/pubs/SchemaCentricCanonicalization-20050523.htm

Deleted: 20020710

Latest version:

http://uddi.org/pubs/SchemaCentricCanonicalization.htm

Previous version:

http://uddi.org/pubs/SchemaCentricCanonicalization-20020710.htm

Author:

Bob Atkinson, Microsoft, bobatk@microsoft.com

Editors:

Selim Aissi, Intel, selim.aissi@intel.com

Maryann Hondo, IBM, mhondo@us.ibm.com

Andrew Hately, IBM, hately@us.ibm.com

Abstract:

Existing XML canonicalization algorithms such as Canonical XML and Exclusive XML Canonicalization suffer from several limitations and design artifacts (enumerated herein) which significantly limit their utility in many XML applications, particularly those which validate and process XML data according to the rules of and flexibilities afforded by XML Schema. The Schema Centric Canonicalization algorithm addresses these concerns.

Status:

This specification has attained the status of Committee Specification. This document is updated periodically on no particular schedule.

Committee members should send comments on this Committee Specification to the uddi-spec@lists.oasis-open.org list. Others should subscribe to and send comments to the uddi-spec-comment@lists.oasis-open.org list. To subscribe, send an email message to mailto:uddi-spec-comment-request@lists.oasis-open.org?body=subscribe with the word "subscribe" as the body of the message.

For information on whether any intellectual property claims have been disclosed that may be essential to implementing this Committee Specification, and any offers of licensing terms, please refer to the Intellectual Property Rights section of the UDDI Spec TC web page (http://www.oasis-open.org/committees/uddi-spec/).

Copyrights:

Table of Contents

1. Introduction

The design of the XML-Signature Syntax and Processing specification requires the execution of a canonicalization algorithm as part of the signature generation process. To date, two different (but closely related) canonicalization algorithms have been broadly proposed:

- Canonical XML, a product of a joint effort between the W3C and the IETF, and
- Exclusive XML Canonicalization, a W3C effort that adapts Canonical XML to modify its treatment of xml:lang attributes, xml:space attributes, and namespace nodes in order to address issues encountered in re-enveloping a signed subdocument.

1.1 Limitations of Existing Canonicalization Algorithms

Both of these algorithms (collectively "the existing algorithms") share some characteristics which cause problems, some considerable, to applications considering their use:

1. The presence of a DTD that validates the XML subdocument being canonicalized is assumed. In particular, default attributes specified in the DTD are included in the output of the canonicalization process.

   With the advent of XML Schema, it is in fact now increasingly rare to find XML documents for which validation is accomplished using a DTD, or, indeed, due to the weak expressiveness of DTDs, to find XML documents for which a DTD which describes the content models of the elements of the document (instead of merely defining entities and the like) can in fact ever be constructed. Thus, the existing algorithms are becoming less and less useful to practical applications of XML.

2. Contrary to the intent of the Namespaces in XML Recommendation, XML documents are *not* canonicalized with respect to the XML namespace prefixes they use. That is, XML documents that are identical except for their choice of namespace prefixes canonicalize to different results under the existing algorithms. Since namespace declarations can appear on *any* element, the need for their preservation can at times be a very significant implementation burden.
3. Canonical XML contains a (pragmatically minor) security hole having to do with how it processes certain esoteric node-sets. Consider a node set which consists of just a single attribute node, one that explicitly references a namespace by use of a namespace prefix. While it is true in Canonical XML that an element node that is not in the node-set still has its namespace axis processed, the rule in Canonical XML (see §2.3) for processing that namespace axis states that only "namespace nodes in the axis *and in the node-set*" (*emphasis* added) are in fact processed. Thus, the canonical representation of our single-attribute-node node-set consists of the processing of *only* the attribute node itself; no namespace attributes are included. Thus, two such single-attribute node-sets whose attributes are character-wise identical but use completely different namespaces as the binding of their prefix will canonicalize to the same result, and that presents a security hole, particularly in applications to digital signatures. Analogous security holes exist with similar node-sets. Whether the same security hole exists in Exclusive XML Canonicalization is likely the case but is not entirely clear.

4. The goal of the existing canonicalization algorithms is to canonicalize an XML subdocument with respect to the liberties of its physical representation permitted within ***only*** the XML 1.0 Recommendation and the Namespaces in XML Recommendation.

   The XML Schema Recommendation permits a considerable number of additional liberties of representation, including (but not limited to) the following:

   a. the optional presence of both comments and processing instructions at completely arbitrary points in the input XML
   b. normalization of whitespace in certain element content (in a like manner as but in addition to the normalization of whitespace within *attributes* mandated by XML 1.0)
   c. the permitted presence of whitespace with no semantic impact imparted by the presence thereof in the content of elements of complex type which have a {content type} of *element-only* (that is, between end-tags and start-tags of elements which are children of such elements)
   d. the specification in a schema of the default value of attributes, which consequently permits without semantic impact their omission in a corresponding XML instance
   e. the specification in a schema of the default value of the content of elements, which operates in a manner similar to that of the specification of the default value of attributes
   f. the inclusion of xsi:schemaLocation and xsi:noNamespaceSchemaLocation attributes as useful hints to the XML Schema processing system but which are not of semantic significance to XML Schema instance itself
   g. within the content of an element of complex type which has a {content type} of *element-only,* the semantic insensitivity to the order within a sequence of elements that validates against a model group whose compositor is *all.* (In contrast, when such occurs within a {content type} of *mixed,* and so there may be non-whitespace interspersed between these elements, the elements may not reasonably be reordered, as their relationship to such characters may have semantic significance to applications.)

h. variability in the lexical representation of the data types built-in to XML Schema and extensions or restrictions thereof, including
   i. the permitted use of any of {true, false, 0, 1} for data of type boolean
   ii. the optional use of leading "+" signs in positive values, and the optional use of leading and trailing zeros in data of type decimal and restrictions thereof, including integer, long, int, nonNegativeInteger, and so on (as well as, of course, user-defined extensions and restrictions)
   iii. for data of type float and double, the use of upper or lower case "e" in scientific notation, the use of leading zeros in the exponent thereof, the use of leading "+" signs on positive values, the use of trailing zeros in the mantissa, and the unnecessary use of leading zeros in the mantissa.
   iv. the permitted use of various time zones to represent the same time value in data of type dateTime and time, as well as two representations for midnight for such data
   v. the permitted use of both upper and lower case in data of type hexBinary
   vi. in data of type base64Binary, the permitted use (per the clarification in the errata to XML Schema of the lexical forms of base64Binary data) of whitespace characters

It should be noted that for these six data types, XML Schema Datatypes does in fact normatively define a corresponding canonical lexical representation. For example, the canonical lexical representation of boolean permits only the use of values in the set {true, false}. However, XML Schema makes use of this canonicalization only in certain circumstances, such as the interpretation of default values of attributes and elements.

There are further data type canonicalization issues which appear to have been overlooked by XML Schema Datatypes:

   vii. (minor) It is not precisely clear from the XML Schema Datatypes specification whether leading zeros are permitted in instances of gYearMonth and gYear when (the absolute value of) the year in question is outside the range of 0001 to 9999. However, in the otherwise analogous passage of the specification of dateTime, such ambiguity is not present (such leading zeros are prohibited), and a reasonable interpretation in these other two cases is to straightforwardly follow that precedent.

viii. the use of mixed case language-tags in data of type language; this is permitted per section "2. The language tag" of RFC 1766, which is (ultimately) the referenced normative specification for the value space of language. (**Note**: this same value space is used by the xml:lang attribute as defined by the XML 1.0 Recommendation; thus, the omission of the canonicalization of the case of xml:lang attributes should reasonably be considered a flaw in even the existing canonicalization algorithms.)

ix. More generally, it is often the case in real-world schemas that various string-valued attributes and elements defined therein are interpreted at the application level as being case-insensitive. This should be capable of being captured by the canonicalization algorithm; were it not, then applications may be forced to remember the exact case used for certain data, a requirement in tension with the application semantic, and quite possibly thus a significant implementation burden.

1.2 Canonicalization Algorithms & Web Services Applications

That these limitations are indeed considerably problematic can be more readily appreciated by considering the implications to certain types of application. One increasingly common and important application of XML is that of so-called "web services". For our purposes here, web services can be thought of as networked applications where the payloads conveyed between network nodes are XML documents, often SOAP requests or responses, which in turn have XML subdocuments in their headers and body. It is observed to be the case that, almost universally, the specification of what constitutes correct and appropriate XML in such circumstances is accomplished using XML Schema.

On the server side of web service applications, it is very often the case that the semantic information conveyed by a request needs to be decomposed, analyzed, and persistently stored, often making use of an underlying relational database to do so. To the extent that such a database is used for storage and indexing purposes, this database gets populated from data received in the body of XML "update" requests. Such population is carried out by "shredding" the semantic information of the XML into a corresponding representation in relational form, losing thereafter the history of that information as having originated in an XML form. Conversely, XML "get" requests are serviced by performing relational operations against the database, then forming an appropriate XML response based on the retrieved data and the schema to which the response must conform.

Certain web service applications will wish to support the use of digital signatures on content which is manipulated by the application. In order to reasonably support such usage, and, in particular, in order to continue to reasonably allow for the shredding of data into an underlying relational store, the signatures in question need to be canonicalized with respect to the full range of liberties of representation afforded by XML Schema. In particular, the problems with the existing algorithms enumerated in the previous section cause especially difficult implementation conundrums in these situations.

The Schema Centric Canonicalization Algorithm is intended to address these concerns.

2. Overview of Schema Centric Canonicalization

The Schema Centric Canonicalization algorithm is intended to be complementary in a hand-in-glove manner to the processing of XML documents as carried out by the assessment of schema validity by XML Schema, canonicalizing its input XML instance with respect to *all* those representational liberties which are permitted thereunder. Moreover, the specification of Schema Centric Canonicalization heavily exploits the details and specification of the XML Schema validity-assessment algorithm itself.

In XML Schema, the analysis of an XML instance document requires that the document be modeled at the abstract level of an information set as defined in the XML Information Set recommendation. Briefly, an XML document's information set consists of a number of ***information items*** connected in a graph. An information item is an abstract description of some part of an XML document: each information item has a set of associated named ***properties***. By tradition, infoset property names are denoted in square brackets, [thus]. There are eleven different types of information items:

1. element information items,
2. attribute information items,
3. comment information items,
4. namespace information items,
5. character information items,
6. document information items,
7. processing instruction information items,
8. unexpanded entity reference information items,
9. document type declaration information items,
10. unparsed entity information items, and
11. notation information items.

Properties on each of these items, for example the [children] property of element information items, connect together items of different types in an intuitive and straightforward way.

The representation of an XML document as an *infoset* lies in contrast to its representation as a *node-set* as defined in XPath. The two notions are conceptually quite similar, but they are not isomorphic. For a given node-set it is possible to construct a semantically equivalent infoset without loss of information; however, the converse is not generally possible. It is the infoset abstraction which is the foundation of XML Schema, and it is therefore the infoset abstraction we use here as the foundation on which to construct Schema Centric Canonicalization algorithm.

The Schema Centric Canonicalization algorithm consists of a series of steps: creation of the input as an infoset, character model normalization, processing by XML-Schema assessment, additional infoset transformation, and serialization.

2.1 Algorithm Input

As was mentioned, the algorithm requires that the data it is to process be manifest as an infoset. If such is not provided directly as input, the data provided must be converted thereto. Two mechanisms for carrying out this conversion are defined:

1. If an octet stream is provided, then it is to be converted into an infoset according to the definition in [XML-Infoset] of the information set which results from the parsing of an XML document represented as an octet stream.
2. If an XPath node-set is provided, then it is to be converted into an infoset according to the rules defined below in this specification.

In addition to the data itself, the canonicalization process requires the availability of appropriate XML Schemas and an indication of the relevant components thereof to which the data purports to conform. In order to be able to successfully execute the canonicalization algorithm, all the data must be valid with respect to these components; data which is not valid cannot be canonicalized.

2.2 Character Model Normalization

The Unicode Standard allows diverse representations of certain "precomposed characters" (a simple example is "ç"). Thus two XML documents with content that is equivalent for the purposes of most applications may contain differing character sequences. However, a normalized form of such representations is also defined by the Unicode Standard.

Schema Centric Canonicalization requires that both its input infoset and all the schema components processed by the XML Schema-Assessment process be transformed as necessary so that all string-valued properties and all sequences of character information items therein be normalized into the Unicode Normalization Form C.

2.3 Processing by XML Schema-Assessment

The third step of the Schema Centric Canonicalization requires that the input infoset be transformed into the so-called "post-schema-validation infoset" (the "PSVI") in the manner defined by the XML Schema Structures recommendation, amended as set forth below. In XML Schema, as the schema assessment process is carried out, the input infoset is augmented by the addition of new properties which record in the information items various pieces of knowledge which the assessment process has been able to infer. For example, attribute information items are augmented with a [schema normalized value] property which contains the result of, among other things, the application of the appropriate schema-specified default-value to the attribute information item (the full list of such augmentation is tabulated in the appendix to XML Schema Structures).

2.4 Additional Infoset Transformation

The PSVI output from XML Schema is next further transformed into what we define here as the "schema-canonicalized infoset" by rules of this specification that are designed to address a few remaining canonicalization issues:

1. the existence of information items in the info set which are completely ignored by the schema assessment process.
2. the existence of the semantically important use of XML namespace prefixes in various embedded languages which are contained strings of the input. For example, an attribute might in fact represent an XPath expression which may internally refer to contextual namespace prefixes. This issue is discussed at some length in Canonical XML. In that specification a decision was made to not canonicalize with respect to namespace prefixes due to the existence of such embedded languages, leaving the output of the algorithm sensitive to the particular prefixes used in the input. Here we choose otherwise, and provide a means by which the algorithm is desensitized to the use of namespace prefixes in embedded languages.
3. the namespaces which, in fact, are used in the output need to be canonicalized with respect to the namespace prefix declaration used for a given such namespace. The overall result is that the output of the Schema Centric Canonicalization algorithm is in no way sensitive to the particular choice of namespace prefixes in its inputs.
4. the previously-mentioned permitted variability in the representation of simple data types in XML Schema

2.5 Serialization of the Schema-Canonicalized Infoset

Finally, the schema-canonicalized infoset is serialized into an XML text representation in a canonical manner, and this serialization forms the output of the algorithm.

The output of the Schema Centric Canonicalization algorithm whose input is the infoset of an entire XML document is well-formed XML. However, if some items in the infoset are logically omitted (that is, their [omitted] property is true), then the output may or may not be well-formed XML, depending on exactly which items are omitted (consider, for example, omitting some element information items but retaining their attributes). However, since the canonical form may be subject to further XML processing, most infosets provided for canonicalization will be designed to produce a canonical form that is a well-formed XML document or external general parsed entity. Note that the Schema Centric Canonicalization algorithm shares these issues of well-formedness of output with the existing canonicalization algorithms.

In such cases where the output of the Schema Centric Canonicalization algorithm is well-formed, then the canonicalization process is idempotent: if *x* is the input infoset, and C represents the application of the Schema Centric Canonicalization algorithm, then *C(x)* is identical to *C(C(x))*. Moreover, in such cases *C(x)* is valid with respect to the same schema component(s) as is *x* (modulo the character sequence length issue noted in the next section).

2.6 Limitations

The Schema Centric Canonicalization algorithm suffers from some of the limitations of Canonical XML. Specifically, as in Canonical XML, the [base URI] of the infoset has no representation in the canonicalized representation, the consequences of which are as in Canonical XML. However, unlike Canonical XML, Schema Centric Canonicalization does not suffer from the loss of notations and external unparsed entity references (these are canonicalized and preserved) nor from the loss of the typing of data (since, in XML Schema, the association of a schema with an XML instance is outside the scope of the specification and therefore is (trivially) preserved by the Schema Centric Canonicalization algorithm).

As in Exclusive XML Canonicalization, the XML being canonicalized may semantically depend on the effect of xml namespace attributes, such as xml:lang and xml:space. As was the case in Exclusive XML Canonicalization, to avoid problems due to the importation of such attributes from information items which are omitted from the canonicalized output, either they must be explicitly given in the apex nodes of the XML information items being canonicalized or they must always be declared with an equivalent value in every context in which the XML information items will be interpreted.

Schema Centric Canonicalization REQUIRES the identification and availability of a relevant schema for the information items which are to be canonicalized. Therefore, information items which lack such schema cannot be canonicalized with this algorithm.

Schema Centric Canonicalization suffers (but arguably in a minor way) from the fact that XML schema-assessment is not strictly speaking deterministic: when an element or attribute information item is validated against a wildcard whose {process contents} property is *lax*, the exact schema-assessment processing of the item which takes place depends on whether "the item, or any items among its [children] if it's an element information item, has a uniquely determined declaration available", where the term "available" here provides a degree of discretion to the validating application and thus a degree of non-determinism to the schema-assessment process. Because Schema Centric Canonicalization makes integral use of the information garnered during schema-assessment, if an item has been *skip*ped due to a wildcard with a {process contents} of *lax* or *skip,* the output of the algorithm for that item must necessarily be different than if the item has not been *skip*ped. Thus, the non-determinism caused by *lax* results directly in non-determinism of the output of the algorithm. In order to reduce the actual occurrence of this non-determinism, to the extent that it does not conflict with other design requirements, it is RECOMMENDED that schemas intended for use with canonicalization avoid the use of a {process contents} of *lax* in their definitions.

In the canonicalized form, the lengths of certain sequences of character information items may differ from that which was input to the algorithm, due to both processing by Unicode character model normalization and to namespace attribute normalization (the latter only occurs for expressions written in embedded languages such as XPath). This length adjustment can in certain circumstances affect the validity of the altered data, and can affect the ability to reference into the data with XPointer character-points and ranges.

3. Specification of Schema Centric Canonicalization

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119.

The specification of the Schema Centric Canonicalization algorithm defines a few items residing in an XML namespace known as the Schema Centric Canonicalization algorithm namespace. The URI of this namespace is:

urn:uddi-org:schemaCentricC14N:2002-07-10

A (non-normative) XML Schema .xsd file containing definitions of the members of this namespace defined by this specification can be found at:

http://www.uddi.org/schema/SchemaCentricCanonicalization.xsd

It should be clearly understood that all the details of the present document are a matter solely of the *specification* of the behavior of the Schema Centric Canonicalization algorithm, not its *implementation*. Implementations are (of course) free to pursue any course of implementation they choose so long as in all cases the output they yield for a given input corresponds exactly to that as is indicated herein. At times the details and language in this specification may have been optimized to attempt to make the presentation and specification more clear and straightforward perhaps at the performance expense of an implementation that were to robotically follow the literal wording thereof. In this regard, attention is specifically drawn to the connection of the this specification with the details of the specification of the XML Schema validity-assessment, the PSVI augmentation process, and the augmentation of the PSVI found in §3.3: depending on the existing software artifacts and other resources upon which they can rely, good implementations are likely to significantly optimize their treatment of these matters especially.

3.1 Creation of Input as an Infoset

The Schema Centric Canonicalization algorithm manipulates the semantic information of an XML instance as represented in the form of an XML Information Set. As such, if the input to the algorithm is not already in this form then it must be converted thereto in order for the algorithm to proceed. This document normatively specifies the manner in which this conversion is to be carried out for two such alternative input data-types (other specifications are free to define additional, analogous conversions). These two data-types are exactly those defined by the XML Signature Syntax and Processing recommendation as being the architected data-types for input to a Transform.

As is noted in the XML Information Set recommendation, it is not intrinsically the case that the [in-scope namespaces] property of an element information item in an infoset will be consistent with the [namespace attributes] properties of the item and its ancestors, though this is always true for an information set resulting from parsing an XML document. However, it is REQUIRED that this consistency relationship hold for the infoset input to the Schema Centric Canonicalization algorithm.

3.1.1 Conversion of an Octet Stream to an Infoset

If the input to the canonicalization algorithm is an octet stream, then it is to be converted into an infoset by parsing the octet stream as an XML document in the manner described in the specification of [XML-Infoset].

Note that this is exactly the same conversion process that must be carried out by software attempting to assess the schema validity of XML data according to the XML Schema Structure recommendation.

3.1.2 Conversion of a Node-set to an Infoset

The conversion of a node-set to an infoset is straightforward, if somewhat more lengthy to describe.

A node-set is defined by the XPath recommendation as "an unordered collection of nodes without duplicates." In this context, the term "node" refers to the definition provided in the data model section of the recommendation. In that section, it is noted that XPath operates on an XML document as a tree, and that there are seven types of node that may appear in such trees:

1. root nodes
2. element nodes
3. attribute nodes
4. text nodes
5. namespace nodes
6. processing instruction nodes
7. comment nodes

The nodes in a given node-set must (by construction; that is, rules that would allow otherwise are lacking in XPath) all be nodes from the same underlying tree instance. If *N* is a node-set, then let *T(N)* be this tree, and let *r(T(N))* be the root node of that tree. The conversion process to an infoset first converts *T(N)* into an equivalent infoset *I(T(N))*, then decorates that infoset to denote which information items therein correspond to nodes originally found in *N*.

Conversion of an XPath node-tree to an infoset is defined recursively in terms of the conversion of individual nodes to corresponding information items. Let *n* be an arbitrary XPath node, and let *{n}* be a node-set containing just the node *n*. Let *i* be the function taking a node as input and returning an ordered list of nodes as output which is defined as follows:

1. If *n* is a root node, then *i(n)* is a single ***document information item***, where:
    a. the **[children]** property is the ordered list resulting from the concatenation of the lists of information items

        $i(c_j)$

        , where c_j ranges over the children of *n* in document order, excepting that those children of *n* (if any) contained within the DTD (if one exists; entity declarations, for example, may still usefully be found therein even if XML Schema is used for validation) are excluded.

    b. the **[document element]** property is either
        i. that member of [children] which results from the conversion of the single child of *n* which is an element node, if such is present, or
        ii. no value, if such is not present.
    c. the **[notations]** property has no value.
    d. the **[unparsed entities]** property has no value.
    e. the **[base URI]** property is unknown.
    f. the **[character encoding scheme]** property is unknown.
    g. the **[standalone]** property has no value.
    h. the **[version]** property has no value.
    i. the **[all declarations processed]** property is false.
2. If *n* is an element node, then *i(n)* is a single ***element information item***, where:
    a. the **[namespace name]** property is the result of the function invocation *namespace-uri({n})*
    b. the **[local name]** property is the result of the function invocation *local-name({n})*
    c. the **[prefix]** property is either
        i. the prefix of the QName which is the result of the function invocation *name({n})*, if such result is not the empty string, or
        ii. no value otherwise.

d. the **[children]** property is the ordered list resulting from the concatenation of the lists of information items

$i(c_i)$

, where c_i ranges over the children of *n* in document order

e. the **[attributes]** property is the unordered set whose members are the collective members of the lists of information items

$i(a_j)$

, where a_j ranges over those attribute nodes in *T({n})* whose parent is *n* (note that such attribute nodes are not *children* of n).

f. the **[in-scope namespaces]** property is the unordered set whose members are the collective members of the lists of information items

$i(n_k)$

(which are by construction namespace information items), where n_k ranges over the set of namespace nodes in *T({n})* whose parent is *n* (note such namespace nodes are not *children* of n).

g. the **[namespace attributes]** property is an unordered set of attribute information items computed as follows. Let *Nn* be the set of namespace information items in the [in-scope namespaces] property of *i(n),* and let *Np* be the set of namespace information items in the [in-scope namespaces] property of *i(m)*, where *m* is the [parent] of *n*. For each namespace information item *s* in *Nn* - *Np* (so, each namespace information item newly introduced on *i(n)*)*,* the [namespace attributes] property contains an attribute information item whose properties are as follows:
   i. the [namespace name] property is (per XML Infoset) "http://www.w3.org/2000/xmlns/"
   ii. the [local name] property is the value of the [prefix] property of *s*.

iii. the [prefix] property is "xmlns"
iv. the [normalized value] property is the value of the [namespace name] property of *s*.
v. the remaining properties are as set forth in the attribute node case below.

Conversely, consider each namespace node *s* in *Np* - *Nn* (so, each namespace information item present on the parent but removed on *n*). The specification of XML Namespaces is such that there can be at most one such *s*, and that it represent a declaration of the default namespace, which is then undeclared by element *i(n)*. If such an *s* exists, then the [namespace attributes] property of *i(n)* additionally contains an attribute information item whose properties are as follows:

vi. the [namespace name] property is "http://www.w3.org/2000/xmlns/"
vii. the [local name] property is the empty string
viii. the [prefix] property is "xmlns"
ix. the [normalized value] property is the empty string
x. the remaining properties are as set forth in the attribute node case below.

h. the **[base URI]** property is unknown.
i. the **[parent]** property is the document or element information item in the infoset rooted at *i(r(T({n}))* which contains this information item in its [children] property.

3. If *n* is an attribute node, then *i(n)* is a single ***attribute information item***, where:
   a. the **[namespace name]** property is the result of the function invocation *namespace-uri({n})*
   b. the **[local name]** property is the result of the function invocation *local-name({n})*
   c. the **[prefix]** property is either
      i. the prefix of the QName which is the result of the function invocation *name({n})*, if such result is not the empty string, or
      ii. no value otherwise.
   d. the **[normalized value]** property is the result of the function invocation *string({n})*
   e. the **[specified]** property is unknown.
   f. the **[attribute type]** property is unknown.
   g. the **[references]** property is unknown.

   h. the **[owner element]** property is the element information item in the infoset rooted at *i(r(T({n}))* which contains this information item in its [attributes] property, if any such element exists, or no value otherwise.

4. If *n* is a text node, then *i(n)* is an ordered list of ***character information items***, one character information item c_j corresponding to each character in the result of the function invocation *string({n})*, where
   a. the **[character code]** property of c_j is the ISO 10646 character code of the corresponding *jth* character in the result of the function invocation *string({n}).*
   b. the **[element content whitespace]** property of c_j is
      i. unknown if the character is whitespace, and
      ii. false otherwise.
   c. the **[parent]** property is the element information item in the infoset rooted at *i(r(T({n}))* which contains this information item in its [children] property.
5. If *n* is a namespace node, then *i(n)* is a single ***namespace information item***, where
   a. the **[prefix]** property is the result of the function invocation *local-name({n}),* unless that returns an empty string, in which case the [prefix] property is no value*.* This perhaps unexpected formulation arises from the fact that in XPath, "a namespace node has an expanded-name: the local part is the namespace prefix (this is empty if the namespace node is for the default namespace); the namespace URI is always null."
   b. the **[namespace name]** property is the result of the function invocation *string({n}).*
6. If *n* is a processing instruction node, then *i(n)* is a single ***processing instruction information item***, where
   a. the **[target]** property is the result of the function invocation *local-name({n}).*
   b. the **[content]** property is the result of the function invocation *string({n}).*
   c. the **[base URI]** property is unknown.
   d. the **[notation]** property is unknown.
   e. the **[parent]** property is the document, element, or document type definition information item in the infoset rooted at *i(r(T({n}))* which contains this information item in its [children] property
7. If *n* is a comment node, then *i(n)* is a single ***comment information item***, where

a. the **[content]** property is the result of the function invocation *string({n}).*
b. the **[parent]** property is the document or element information item in the infoset rooted at *i(r(T({n}))* which contains this information item in its [children] property.

Having defined the function *i,* we now return to completing the specification of the details of the node-set to infoset conversion process.

Let *N* be a node-set, and consider the document information item returned by the function invocation *i(r(T(N))*. Define the infoset *I(T(N))* to be that set of information items which are transitively reachable from *i(r(T(N))* through any of the properties defined on any of the information items therein. This infoset represents the conversion of the node tree *T(N)* into a corresponding infoset.

Recall that the node-set *N* is in fact a subset of *T(N)*. This relationship therefore needs to be represented in I(T(N)). To that end, we here define a new boolean infoset property called **[omitted]**. Unless otherwise indicated by some specification, the value of the [omitted] property of any information item is always to be taken to be 'false'. The present specification, however, defines that for all information items in *I(T(N))* the value of [omitted] is 'true' except those items which, for some *n* in *N,* are members of the list returned from *i(n)*.

This completes the specification of the node-set to infoset conversion process.

3.2 Character Model Normalization

The Unicode Standard allows diverse representations of certain "precomposed characters" (a simple example is "ç"). Thus two XML documents with content that is equivalent for the purposes of most applications may contain differing character sequences. However, a normalized form of such representations is also defined by the Unicode Standard.

It is REQUIRED in Schema Centric Canonicalization that both the input infoset provided thereto and all the schema components to processed by the XML Schema-Assessment process used therein be transformed as necessary so that all string-valued properties and all sequences of character information items therein be normalized into the Unicode Normalization Form C as specified by the algorithm defined by the Unicode Standard.

As a (non-normative) note of implementation, in the case where the to-be-canonicalized XML instance and the XML schema specifications thereof are input to the canonicalization process as physical files, this normalization can usually be most straightforwardly accomplished simply by normalizing the characters of these files first before commencing with the remainder of the canonicalization process.

3.3 Processing by XML Schema Assessment

Once the input infoset is normalized with respect to its character model, the Schema Centric Canonicalization algorithm carries out schema assessment by appealing to the third approach listed in §5.2 Assessing Schema-Validity of the XML Schema recommendation and attempting to carry out strict assessment of the element information item which is the value of the [document element] property of the document information item of the infoset.

In XML Schema, as the schema assessment process is carried out, the infoset input to that process is augmented by the addition of new properties which record in the information items various pieces of knowledge which the assessment process has been able to discern. For example, attribute information items are augmented with a [schema normalized value] property which contains the result of, among other things, the application of the appropriate schema-specified default-value to the attribute information item.

The Schema Centric Canonicalization algorithm makes use of this augmentation. Specifically, suppose *I* is the character-normalized version of the infoset which is input to the algorithm, possibly after conversion from another data-type. Then the next step of the algorithm forms the so-called "post-schema-validation infoset" (the "PSVI", or more precisely, *PSVI(I)*) in exactly the manner prescribed as a consequence of the assessment process defined in the XML Schema Structures specification as amended in the manner set forth below. If *PSVI(I)* cannot be so formed, due to, for example, a failure of validation, then the Schema Centric Canonicalization algorithm terminates with a fatal error.

In XML Schema Structures, the augmentation process of schema assessment fails to record a small number of pieces of information which it has learned and which we find crucially necessary to have knowledge of here. Accordingly, the PSVI generation process referred to by this specification is exactly that of the XML Schema Structures recommendation as amended as follows:

3.8.5 Model Group Information Set Contributions

If the schema-validity of an element information item has been assessed as per Element Sequence Valid (§3.8.4) by a model group whose {compositor} is *all*, then in the post-schema-validation infoset it has the following property:

PSVI Contributions for element information items
[validating model group all] An ·item isomorphic· to the model group component involved in such assessment.

3.4 Additional Infoset Transformation

The fourth step of the Schema Centric Canonicalization algorithm further augments and transforms the PSVI to produce the "schema-canonicalized infoset". This involves a pruning step, a namespace prefix desensitization step, a namespace attribute normalization step, and a data-type canonicalization step.

3.4.1 Pruning

Some information items in the PSVI in fact do not actively participate in the schema assessment process of XML Schema. They are either ignored completely by that process, or used in an administrative capacity which is not central to the outcome. Thus, these items need to be pruned from the PSVI in order that they not affect the output of canonicalization. Similarly, declarations of notations and unparsed entities which are not actually referenced in the canonicalized representation should also be removed.

To this end, the [omitted] property is set to 'true' for any information item *info* in the PSVI for which at least one of the following is true:

1. *info* is a (necessarily whitespace) character information item which is a member of the [children] of an element information item whose [type definition] is a complex type

schema component whose {content type} property is *element-only*

2. *info* is an attribute information item whose [namespace name] is identical to "http://www.w3.org/2001/XMLSchema-instance" and whose [local name] is one of "schemaLocation" or "noNamespaceSchemaLocation"
3. *info* is a notation information item for which there does not exist an attribute or element information item in the infoset whose [omitted] property is false, whose [member type definition] (if present) or [type definition] (otherwise) property is either
    a. a NOTATION simple type (or restriction or extension thereof)
    b. a list of same

    and whose [schema normalized value] is identical (in the former case) or contains a list item which is identical (in the later case) to the [name] of the notation information item

4. *info* is an unparsed entity information item for which there does not exist an attribute or element information item in the infoset whose [omitted] property is false, whose [member type definition] (if present) or [type definition] (otherwise) property is either
    a. an ENTITY simple type (or restriction or extension thereof)
    b. a list of same

    and whose [schema normalized value] is identical (in the former case) or contains a list item which is identical (in the later case) to the [name] of the unparsed entity information item

3.4.2 Namespace Prefix Desensitization

The goal of namespace prefix desensitization is to first identify those information items in the infoset which make use of namespace prefixes outside of XML start and end tags (that is, information of type QName and derivations and lists thereof as well as information representing an expression written in some embedded language which makes use of the XML Namespaces specification in a embedded-language-specific manner), and next to annotate the infoset in order to indicate exactly where and in what manner uses of particular XML namespace prefixes in fact occur. That is, desensitization is a two-step process: a data location step, followed by an annotation step.

Note that the notion of embedded language used here includes not only languages (such as XPath) which are represented in XML as the content of certain strings but also those (such as XML Query) which make use of *structured* element content. In all cases, however, in order to be namespace-desensitizeable it is REQUIRED that all references to XML namespace prefixes do in fact ultimately lie in information identified as being of a simple type (usually strings). It is, however, permitted that these prefixes may be found in simple types which are attributes and / or the content of elements perhaps deep in the sub-structure of the element rooting the occurrence of the embedded language.

Moreover, in order to be namespace-desensitizeable, it is REQUIRED that the semantics of each embedded language not be sensitive to the specific namespace prefixes used, or the character-count length thereof: one MUST be permitted to (consistently) rewrite any or all of the prefixes used in an occurrence of a language with arbitrary other (appropriately declared) prefixes, possibly of different length, without affecting the semantic meaning in question.

Each particular embedded language for which namespace desensitization is to be done MUST be identified by a name assigned to it by an appropriate authority. It is REQUIRED that this name be of data-type anyURI. This specification assigns the following URIs as names of particular embedded languages:

URI	**Embedded Language**
http://www.w3.org/TR/1999/REC-xpath-19991116	the embedded language which consists of sequences of characters which conform to the any of the grammatical productions of the XPath 1.0 specification
http://www.w3.org/TR/2001/REC-xmlschema-2-20010502	an embedded language which consists of sequences of characters which are of type QName or derivations and/or lists (and their derivations) thereof

The data location step of desensitization makes use of canonicalization-specific annotations to XML Schema components. It is the case in XML Schema that the XML representation of all schema components allows the presence of attributes qualified with namespace names other than the XML Schema namespace itself; this is manifest in the schema-for-schemas as the presence of an

```
<xs:anyAttribute namespace="##other"
processContents="lax"/>
```

definition in the schema for each of the various schema components. As is specified in XML Schema Structures, such attributes are represented in the infoset representation of the schema inside the {attributes} property of an Annotation schema component which in turn is the value of the {annotation} property of the annotated schema component in question (i.e.: the Annotation is the {annotation} of the Attribute Declaration, the Element Declaration, or whatever). Within the Schema Centric Canonicalization algorithm namespace, we define a couple of attributes intended for use as such annotations to schema components:

1. The **embeddedLang** attribute, which is of type anyURI, is defined in the Schema Centric Canonicalization algorithm namespace. When used as an attribute annotation to a schema component, an embeddedLang attribute indicates that an information item which validates against the schema component in question in fact contains information written in a certain, fixed embedded language whose name is indicated in the value of the embeddedLang attribute. The embeddedLang attribute may also be used within a schema instance (but only, of course, where such attributes are permitted by the corresponding schema); this is in loose analogy to how the xsi:type attribute is used. In such situations, the [owner element] of the embeddedLang attribute in fact contains information written in a certain, fixed embedded language whose name is indicated in the value of the embeddedLang attribute. The use of an embeddedLang attribute in a schema instance supercedes any identification of embedded language that may be provided by its schema.

2. The **embeddedLangAttribute** attribute, which is of type QName, is defined in the Schema Centric Canonicalization algorithm namespace. When used as an attribute annotation to a schema component, an embeddedLangAttribute attribute indicates that an information item which validates against the schema component in question in fact contains information written in an embedded language whose name is dynamically indicated in the information item (necessarily an element information item) as the value of a certain attribute thereof, namely the attribute whose qualified name is indicated in the value of the embeddedLangAttribute attribute.

In order to specify how these attributes are used, we define an auxiliary function in order to model the inheritance of annotations in schemas from types to elements and attributes and from base types to derived types. Let *i* be an information item, *a* be a string (representing the name of an attribute), and *ns* be either a URI (representing the name of an XML namespace) or the value absent. Define the function *getAnnot(i, a, ns)* as follows:

1. If *i* is an element information item, then
    a. If the [element declaration] property of *i* contains in its {annotation} property an Annotation schema component which contains in its {attributes} property an attribute information item whose {name} is *a* and whose {target namespace} is *ns* (that is, if the [element declaration] property of *i* "has an (a,ns) annotation attribute"), then *getAnnot(i, a, ns)* is the value of that attribute
    b. Otherwise, let *t* be the [member type definition] property of *i* (if it exists) or the [type definition] property of *i* (otherwise). Then *getAnnot(i, a, ns)* is *getAnnot(t, a, ns).*
2. If *i* is an attribute information item, then
    a. If the [attribute declaration] property of *i* has an (a,ns) annotation attribute, then *getAnnot(i, a, ns)* is the value of that attribute.
    b. Otherwise, let *t* be the [member type definition] property of *i* (if it exists) or the [type definition] property of *i* (otherwise). Then *getAnnot(i, a, ns)* is *getAnnot(t, a, ns).*
3. If *i* is an information item which is item isomorphic to a complex type definition schema component, then,
    a. If *i* has an (a,ns) annotation attribute, then *getAnnot(i, a, ns)* is the value of that attribute
    b. If the {base type definition} property *t* of *i* is not the ur-type definition, then *getAnnot(i, a, ns)* is *getAnnot(t, a, ns)*
    c. Otherwise, *getAnnot(i, a, ns)* is absent.
4. If *i* is an information item which is item isomorphic to a simple type definition schema component, then,
    a. If *i* has an (a,ns) annotation attribute, then *getAnnot(i, a, ns)* is the value of that attribute.
    b. If the {variety} property of *i* is *atomic*, and if the {base type definition} property *t* of *i* is not the ur-type definition, then *getAnnot(i, a, ns)* is *getAnnot(t, a, ns)*

    c. If the {variety} property of *i* is *list,* then *getAnnot(i, a, ns)* is *getAnnot(t, a, ns),* where *t* is the {item type definition} property of *i*.
    d. Otherwise, *getAnnot(i, a, ns)* is absent.
5. Otherwise, *getAnnot(i, a, ns)* is absent.

The data location step of desensitization is carried out as follows. Let *sccns* be the Schema Centric Canonicalization namespace. Consider in turn each attribute and element information item *x* in the pruned PSVI:

1. If *x* is an element information item, and if the [attributes] of *x* contain an attribute *a* whose [namespace name] is *sccns* and whose [local name] is *"embeddedLang",* then *x* is identified as being associated with the embedded language which is the value of the [schema normalized value] of *a* (if present) or the [normalized value] of *a* (otherwise).
2. Otherwise, if *x* is an element information item, and if *getAnnot(x, "embeddedLangAttribute", sccns)* is not absent, then *x* is identified as being associated with the embedded language which is the [schema normalized value] (if present) or the [normalized value] (otherwise) of the member of the [attributes] of *x* whose name is the value of *getAnnot(x, "embeddedLangAttribute", sccns);* if no such member of [attributes] exists, a fatal error occurs.
3. Otherwise, if *getAnnot(x, "embeddedLang", sccns)* is not absent, then *x* is identified as being associated with the embedded language which is is the value thereof;
4. Otherwise, *x* is not associated with any embedded language by means of the embeddedLang or embeddedLangAttribute attributes, though such an association may be indicated by other means, such as by fiat in some specification.

To that last point this specification REQUIRES that a schema component representing any of the following:

1. the type of the element named "XPath" contained in elements of type dsig:TransformType (where the prefix "dsig" is bound to the XML Signature Syntax and Processing namespace: http://www.w3.org/2000/09/xmldsig#), or
2. the "xpath" attribute whose [owner element] is the element xsd:selector (where the prefix "xsd" is bound to the XML Schema namespace: http://www.w3.org/2001/XMLSchema), or

3. the "xpath" attribute whose [owner element] is the element xsd:field (where the prefix "xsd" is bound as before)

is to be considered by definition as possessing an embeddedLang attribute with value http://www.w3.org/TR/1999/REC-xpath-19991116 in the {attributes} property of its {annotation} property (that is, they are by definition annotated as being XPath 1.0 expressions).
Moreover, any attribute or element information item whose [member type definition] (if present) or [type definition] (otherwise) property is any of:

1. QName or a derivation thereof
2. a list of QName or a derivation thereof
3. a derivation of a list of QName or a derivation thereof

is identified as being associated with the embedded language whose name is http://www.w3.org/TR/2001/REC-xmlschema-2-20010502.

Other specifications are encouraged to provide similar legacy-supporting definitions when appropriate: the price of not identifying an embedded language when one is actually in use is that the canonicalized output will (almost certainly) be non-operational due to dangling or erroneously-bound namespace prefixes.

Following the data location step, the processing of the attribute and element information items identified as being associated with embedded languages is carried out by the annotation step of namespace prefix desensitization in what is necessarily an embedded-language-specific manner. Implementations of the Schema Centric Canonicalization algorithm will need to understand the syntax and perhaps some semantics of each of the embedded languages whose uses they encounter as they carry out canonicalization. Should an embedded language which is not appropriately understood be encountered, the Schema Centric Canonicalization algorithm terminates with a fatal error. All implementations of the Schema Centric Canonicalization algorithm MUST in this sense fully understand the (XPath) embedded language identified as http://www.w3.org/TR/1999/REC-xpath-19991116 as well as the embedded language identified as http://www.w3.org/TR/2001/REC-xmlschema-2-20010502.
In all cases the execution of the annotation step is manifest in the augmented PSVI in a uniform manner. Specifically, let *x* be an attribute or element information item which is identified by the language-specific processing as containing one or more uses of XML namespace prefixes in its [schema normalized value]

property *y*. If any of these uses of XML namespace prefixes in *y* is in a form other than a occurrence of a QName, then a fatal error occurs. Otherwise, *x* is additionally augmented by the language-specific processing with a **[prefix usage locations]** property which contains, corresponding to the sequence of all the QNames in *y*, an ordered sequence of one or more triples *(offset, prefix, namespace URI)* where

1. *offset* is the zero-based offset from the start of *y* of the first character of a QName
2. *prefix* is the string value of the prefix of that QName (not, to be clear, including any trailing colon), if any is present, or no value otherwise.
3. *namespace URI* is the in-scope binding of the that XML namespace prefix (or the default XML namespace, if *prefix* is no value), or no value if no such binding exists (which necessarily must result from a use of the default XML namespace prefix in a context where no declaration for that prefix is in scope),

and these triples occur in increasing order by *offset*.

This concludes the specification of the namespace prefix desensitization step.

3.4.3 Namespace Attribute Normalization

The next step in the series of infoset transformations carried out by the Schema Centric Canonicalization algorithm is that of normalizing the actual XML namespace prefix declarations in use. The XML namespace recommendation allows namespaces to be multiply declared throughout an XML instance, possibly with several and different namespace prefixes used for the same namespace. In the canonical representation, we remove this flexibility, declaring each namespace just as needed, and using a deterministically constructed namespace prefix in such declaration. In this procedure, we borrow heavily from some of the similar work carried out in the Exclusive XML Canonicalization recommendation. We begin with some definitions.

ancestor information item

An *ancestor* information item *a* of an information item *i* in an infoset is any information item transitively reachable from *i* through traversal of the [parent] properties of element, processing instruction, unexpanded entity reference, character, comment, and document type declaration information items, and the [owner element] property of attribute information items. Notation, unparsed entity, and namespace information items have no ancestors, nor do attribute information items which appear in elements other than in their [attributes] properties. Note that the information item *i* is not an ancestor of itself.

self-ancestor

A self-ancestor of an information item is either the information item itself or an ancestor thereof.

output parent

The output parent of an information item *i* in an infoset is (noting that the ancestor relationship is transitive) the *nearest* ancestor of *i* which is an element information item whose [omitted] property is false, or no value if such an ancestor does not exist.

visibly utilize

An element information item *e* in an infoset is said to *visibly utilize* an XML namespace prefix *p* if any of the following is true:

1. the [prefix] property of *e* is identical to *p* (note that this includes the case where both are no value),
2. *e* has a [prefix usage locations] property, and that property value contains some triple whose *prefix* member is identical to *p*
3. there exists an attribute information item *a* in the infoset whose [owner element] property is *e*, whose [omitted] property is false, and either
    a. the [prefix] property of *a* is identical to *p*,
    b. *a* has a [prefix usage locations] property, and that property value contains some triple whose *prefix* member is identical to *p*

The execution of the namespace attribute normalization step adds **[normalized namespace attributes]** properties to certain element information items in the infoset. Let *e* be any element information item whose [omitted] property is false. Then the [normalized namespace attributes] property of *e* is that unordered set of attribute information items defined recursively as follows.

Let *Ne* be the set of all namespace information items *n* in the [in-scope namespaces] property of *e* where *n* is visibly utilized by *e*. Let *NAp* be the set of attribute information items in the [normalized namespace attributes] property of any self-ancestor of *p*, where *p* is the output parent of *e* and if *p* is not no value, or the empty set if no such output parent exists. Let *namespaces(Ne)* be the set of strings consisting of the [namespace name] properties of all members of *Ne*, and let *namespaces(NAp)* be the set of strings consisting of the [normalized value] properties of all members of *NAp.*

For each namespace URI *u* in *namespaces(Ne)* - *namespaces(NAp)* (so, the name of each namespace with a prefix newly utilized at *e*), the [normalized namespace attributes] property of *e* contains an attribute information item whose properties are as follows:

1. the [namespace name] property is (per XML Infoset) "http://www.w3.org/2000/xmlns/"
2. the [local name] property is either:
   a. a the string "xml" if the namespace value is "http://www.w3.org/XML/1998/namespace"
   b. a string of the form "n" concatenated the canonical lexical representation of a non-negative integer *i* (for example "n0", "n1", "n2", and so on) where the particular integer *i* in question is chosen as described just below.
3. the [prefix] property is "xmlns"
4. the [normalized value] property is the value *u*.
5. the [schema normalized value] property is identical to the [normalized value] property
6. the remaining properties are as set forth above in the specification of conversion of attribute nodes to information items.

Deleted:

XML namespace prefixes used in the [normalized namespace attributes] property (which are manifest in the [local name] properties of the attribute information items contained therein) are chosen as follows. Let *e* be any element containing a [normalized namespace attributes] property. Let *l* be the ordered list resulting from sorting the [normalized namespace attributes] property of *e* according to the *sort* function described below. Let *k* be the maximum over all the ancestors *a* of *e* of the integers used per (b) above to form the [local name] property of any attribute item in the [normalized namespace attributes] property of *a,* or -1 if no such attribute items exist. Then the attributes of *l*, considered in order, use, in order, the integers *k+1, k+2, k+3*, and so on in the generation of their [local name] as per (b) above, excepting only that if *wildcardOutputRoot(e)* is *true*, then (in order to avoid collisions) any integer which would result in a [local name] property which was the same as the [prefix] property of some namespace item in the [in-scope namespaces] property of *e* is skipped.

Now that the declaration of necessary namespace attributes has been successfully normalized (and, canonically, the default namespace has been left undeclared), we apply these declarations in the appropriate places by defining appropriate **[normalized prefix]** and **[prefix & schema normalized value]** properties. Let *info* be any information item in the infoset whose [omitted] property is false. Then:

1. If *info* is an element or attribute information item whose [namespace name] property has no value, then the [normalized prefix] property of *info* exists but is no value.
2. If *info* is an element or attribute information item whose [namespace name] property is not no value, then let *a* be that namespace declaration attribute in the [normalized namespace attributes] of some self-ancestor of *info* where the [normalized value] property of *a* is identical to the [namespace name] property of *info* (if no such *a* exists, a fatal error occurs. This can occur, for example, if all element information items in the infoset are omitted, but some attributes are retained.). The [normalized prefix] property of *info* then exists and is the [local name] property of *a*.

Moreover, if *info* contains a [prefix usage locations] property, then *info* also contains a [prefix & schema normalized value] property which is identical to the [schema normalized value] property of *info* except for differences formed according to the following procedure. Consider in turn each triple *t* found in the [prefix usage locations] property of *info*. Let *normalizedPrefixUse(t)* be those characters of the [prefix & schema normalized value] property of *info* which correspond to the characters of the [schema normalized value] property of *info* whose zero-based character-offsets lie in the semi-open interval [*offset*, *offset+cch-1+z*), where

1. *offset* is the *offset* member of *t*,
2. *cch* is the number of characters in the *prefix* member of *t* (if *prefix* is not no value) or zero (otherwise*),* and
3. *z* is one if *prefix* is not no value and the offset+cch-1+1'st character of the [schema normalized value] of *info* property is a colon, and zero otherwise.

Then the characters of *normalizedPrefixUse(t)* are determined as follows:

1. If the *namespace URI* of *t* has no value, then *normalizedPrefixUse(t)* is the empty string.
2. Otherwise, let *a* be that namespace declaration attribute in the [normalized namespace attributes] of some self-ancestor of *info* where the [normalized value] property of *a*

is identical to the *namespace URI* of *t* (if no such *a* exists, a fatal error occurs). Then *normalizedPrefixUse(t)* is the [local name] of *a* followed by a colon.

This completes the specification of the namespace attribute normalization step.

3.4.4 Data-type Canonicalization

The XML Schema Datatypes specification defines for a certain set of its built-in data-types a canonical lexical representation of the values of each of those data types. To that identified set of canonical representations Schema Centric Canonicalization adds several new rules; in some cases, it refines those rules provided by XML Schema.

The most complicated part of data type canonicalization lies in dealing with character sequences which are as a matter of application-level schema design considered to be case insensitive. It is important that case-insensitivity of application data be integrated into the canonicalization algorithm: if it were not, then applications may be forced to remember the exact case used for certain data when they otherwise would not need to, a requirement which may well be in tension with the application semantic of case-insensitivity, and thus quite possibly a significant implementation burden.

The relevant technical reference for case-mapping considerations for Unicode characters is a technical report published by the Unicode Consortium. Case-mapping of Unicode characters is more subtle than readers might naively intuit from their personal experience. The mapping process can at times be both locale-specific (Turkish has special considerations, for example) and context-dependent (some characters case-map differently according to whether they lie at the end of a word or not). Mapping of case can change the length of a character sequence. Upper and lower cases are not precise duals: there exist pairs of strings which are equivalent in their upper case-mapping but not in their lower case, and visa versa.

In order to accommodate these flexibilities, we define several attributes within the Schema Centric Canonicalization algorithm namespace in order to assist with the identification of data which is to be considered case-insensitive and the precise manner in which that is to be carried out. As was the case for the embeddedLang and embeddedLangAttribute attributes previously defined, these attributes are intended to be used as annotations of relevant schema components.

The **caseMap** attribute, which is of type language, is defined in the Schema Centric Canonicalization algorithm namespace. When used as an attribute annotation to a schema component, a caseMap attribute indicates that case-mapping is to be performed on data which validates against the schema component according to the case-mapping rules of the fixed locale identified by the value of the attribute.

The **caseMapAttribute** attribute, which is of type QName, is defined in the Schema Centric Canonicalization algorithm namespace. When used as an attribute annotation to a schema component, a caseMapAttribute attribute indicates that an information item which validates against the schema component in question is to be case mapped during the canonicalization process according to the rules of the locale which is dynamically indicated in the information item (necessarily an element information item) as the value of a certain attribute thereof, namely the attribute whose qualified name is indicated in the value of the caseMapAttribute attribute (which must be of type language or a restriction thereof).

The **caseMapKind** attribute, which is of type string but restricted to the enumerated values "upper", "lower", and "fold", is defined in the Schema Centric Canonicalization algorithm namespace. When used as an attribute annotation to a schema component, a caseMapKind attribute indicates whether upper-case or lower-case mapping or case-folding is to be carried out as part of the canonicalization process. If this attribute is contextually absent but at least one of caseMap or caseMapAttribute is contextually present, upper-case mapping is carried out.

Traditional ASCII-like case insensitivity can be most easily approximated by simply specifying "fold" for the caseMapKind attribute and omitting both caseMap and caseMapAttribute. Also, schema designers are cautioned to be careful in combining case-mapping annotations together with length-limiting facets of strings and URIs, due to the length-adjustment that may occur during canonicalization.

The data-type canonicalization step of Schema Centric Canonicalization is carried out according to the following rules:

1. Per the relevant clarification E2-9 in the errata to XML Schema, the canonical lexical representation of a datum of type base64Binary must conform to the grammatical production *Canonical-base64Binary* as defined therein. That production permits in the representation only valid base64 encodings which *only* contain characters from the base64 alphabet as defined by section "6.8 Base64 Content-Transfer-Encoding" of RFC 2045 (in particular, whitespace characters are not in the alphabet), excepting only that the representation is to be formed into lines of exactly 76 characters (except for the last line, which must be 76 characters or shorter) by the appropriate periodic occurrence of a line-feed character (that is, the character whose character code is (decimal) 10) at the end of each line (including the last).
2. The canonical lexical representation of a datum of type dateTime permits only the lexical representation 00:00:00 to denote a time value of midnight (that is, the representation 24:00:00 is prohibited). Further (per XML Schema) either the time zone must be omitted or, if present, the time zone must be Coordinated Universal Time (UTC) indicated by a "Z".
3. The canonical lexical representation of a datum of type float or double is defined by prohibiting certain options from the lexical representation. Specifically, the exponent must be indicated by "E". Leading zeroes and the preceding optional "+" sign are prohibited in the exponent. For the mantissa, the preceding optional "+" sign is prohibited and the decimal point is required. For the exponent, the preceding optional "+" sign is prohibited. Leading and trailing zeroes are prohibited subject to the following: number representations must be normalized such that there is a single digit to the

left of the decimal point and at least a single digit to the right of the decimal point such that the number of of leading zeros in the overall sequence of such digits is a small as otherwise possible.

4. The canonical lexical representation of a datum of type language permits only the use of upper case characters.
5. The canonical lexical representation of a datum of type gYearMonth and gYear prohibits the use of leading zeros for values where the absolute value of the year in question is outside the range of 0001 to 9999.
6. The canonical lexical representation of an element or attribute information item *info* which of type string or anyUri or a restriction thereof and where either of the following is true:
    a. the following is true
        i. *getAnnot(info, "caseMap", sccns)* is present, or, if not
        ii. *getAnnot(info, "caseMapAttribute", sccns)* is present
    b. *getAnnot(info, "caseMapKind", sccns)* is present

    is the result of the application of the function *caseMap* with the parameters

    c. the sequence of characters comprising the value of the element or attribute in question,
    d. the language indicated according to the applicable case i. or ii. above, if any, or the value absent otherwise,
    e. *getAnnot(info, "caseMapKind", sccns).*
7. If none of the preceding rules apply, the canonical lexical representation of a datum of primitive type for which XML Schema Datatypes defines a canonical lexical representation is the representation defined therein.
8. If none of the preceding rules apply, the canonical lexical representation of a datum which is of a primitive type is the not-further-processed representation of the datum itself.
9. The canonical lexical representation of a datum of a type which is derived by list is that which is defined by the XML Schema Datatypes specification (note that this includes the collapsing of the whitespace therein).
10. If none of the preceding rules apply, the canonical lexical representation of a datum which is of a simple type that is a restriction of a type for which a canonical lexical representation is defined is the representation of the datum

according to the canonical lexical representation so defined for that base type.

Thus, a canonical lexical representation for all non-union simple types is defined.

The function *caseMap* takes three input parameters:

1. a sequence of characters whose case is to be mapped,
2. a locale in the form of a language in whose context the mapping is to be carried out, or the value absent, which is to be treated as if "en" were provided,
3. either the string "upper", the string "lower", the string "fold", or the value absent, indicating whether upper-case or lower-case mapping or case-folding is to be carried out; the value absent is treated as if "upper" were provided.

The upper-case or lower-case mapping process of the *caseMap* function is carried out in the context of the indicated locale according to the (respectively) UCD_upper() or UCD_lower() functions as specified by the Unicode Consortium. The case-folding process is carried out by mapping characters through the CaseFolding.txt file in the Unicode Character Database as specified by the Unicode Consortium.

To carry out the data-type canonicalization step in the Schema Centric Canonicalization algorithm, the [schema normalized value] property of all element and attribute information items in the output of the namespace attribute normalization step whose [member type definition] (if present) or [type definition] (otherwise) property is a simple type is replaced by the defined canonical lexical representation of the member of the relevant value space which is represented by the [schema normalized value].

The infoset which is output from the data-type canonicalization step is the schema-canonicalized infoset.

3.5 Serialization of the Schema-Canonicalized Infoset

The final step in the Schema Centric Canonicalization algorithm is the serialization of the schema-canonicalized infoset into a sequence of octets.

In the description of the serialization algorithm that follows, at various times a statement is made to the effect that a certain sequence of characters is to be emitted or output. In all cases, it is to be understood that the actual octet sequences emitted are the corresponding UTF-8 representations of the characters in question. The character referred to as "space" has a character code of (decimal) 32, the character referred to as "colon" has a character code of (decimal) 58, and the character referred to as "quote" has a character code of (decimal) 34.

Also, the algorithm description makes use of the notation "*info[propertyName]*". This is to be understood to represent the value of the property whose name is *propertyName* on the information item *info*.

The serialization of the schema-canonicalized infoset, and thus the output of the overall Schema Centric Canonicalization algorithm, is defined to be the octet sequence that results from the function invocation *serialize(d)*, where *d* is the document information item in the schema-canonicalized infoset, and *serialize* is the function defined below.

3.5.1 The function serialize

The *serialize* function is defined recursively in terms of the serialization of individual types of information item. Let the functions *recurse, sort, escape, wildcarded,* and *wildcardOutputRoot* be defined as set forth later. Let *info* be an arbitrary information item. Let *serialize* be the function taking an information item as input and returning an sequence of octets as output which is defined as follows.

1. If *info* is a **document information item**, then *serialize(info)* is the in-order concatenation of the following:
    a. if info[omitted] is false, and if either info[notations] or info[unparsed entities] contains a notation or an unparsed entity information item (respectively) whose [omitted] property is false, then
        i. the characters "<!DOCTYPE "
        ii. the appropriate case from the following
            1. if *wildcarded(info[document element])* is false, then if info[document element][normalized prefix] is not no value, then the characters thereof, followed by a colon
            2. if *wildcarded(info[document element])* is true, then if info[document element][prefix] is not no value, then the characters thereof, followed by a colon

        iii. the characters of info[document element][local name]
        iv. the characters " ["
        v. recurse(sort(info[notations]))
        vi. recurse(sort(info[unparsed entities]))
        vii. the characters "]>"
    b. recurse(info[children])

2. If *info* is an **element information item**, then *serialize(info)* is:
    a. if info[validation attempted] is *full* or *partial* and info[validity] is not *valid*, then a fatal error occurs.
    b. otherwise, the in-order concatenation of the following:
        i. if info[omitted] is false, then
            1. the character "<"
            2. the appropriate case from the following:
                a. if *wildcarded(info)* is false, then if info[normalized prefix] is not no value, then the characters thereof, followed by a colon
                b. if *wildcarded(info)* is true, then if info[prefix] is not no value, then the characters thereof, followed by a colon
            3. the characters of info[local name]
            4. if info[normalized namespace attributes] exists, then recurse(sort(info[normalized namespace attributes]))
            5. if *wildcardOutputRoot(info)* is true, then recurse(sort(*N*)), where *N* is info[in-scope namespaces] but with the item therein having the prefix "xml" removed.
            6. if *wildcarded(info)* is true and *wildcardOutputRoot(info)* is false, then recurse(sort(info[namespace attributes])).
        ii. recurse(sort(info[attributes]))
        iii. if info[omitted] is false, then
            1. the character ">"
        iv. the appropriate case from the following:
            1. if the property info[prefix & schema normalized value] is present, then
                a. if info[children] contains any character information item *c*

        where *c*[omitted] is true, then the empty octet sequence,
    b. otherwise, escape(info[prefix & schema normalized value])
2. else if the property info[schema normalized value] is present, then
    a. if info[children] contains any character information item *c* where *c*[omitted] is true, then the empty octet sequence,
    b. otherwise, escape(info[schema normalized value]),
3. else if at least one member of info[children] is an element information item which possesses a [validating model group all] property, then let the subsequence of info[children] consisting of all those elements which possess a [validating model group all] property be partitioned into into *k* subsequences l_1 to l_k such that *k* is as small as possible and all items of a given subsequence share the same model group information item for their [validating model group all] property (XML Schema assures that this is well-defined), and let *children'* be a re-ordering of info[children] according to the following constraints:
    a. if an item *c* of info[children] possesses a [validating model group all] property, and is therefore contained in subsequence l_i for some *i*, then the relative order of *c* in *children'* with respect to
        i. any item *d* of l_i different than *c* is the same as the relative ordering of *c* and *d* in *sort*(l_i)
        ii. any item *e* of l_j (for some $i \neq j$) is the same as the relative ordering of the first items of l_i and l_j
        iii. any other item *f* of info[children] is the same

as the relative ordering in info[children] of *f* with that item *g* of l_i where the index of *g* in l_i is the same as the index of *c* in *sort*(l_i)

b. if items *m* and *n* of info[children] do not posses a [validating model group all] property, then they occur in *children'* in the same relative order as they occur as items in info[children]

then, recurse(children')

4. otherwise, if info[content type] is element-only, then recurse(nwsChildren), where nwsChildren is the result of removing from info[children] those character information items whose [character code] is defined as a white space in the XML 1.0 Recommendation (this reflects the validation rule in clause 2.3 of §3.4.4 of XML Schema).
5. otherwise, recurse(info[children])

v. if info[omitted] is false, then
   1. the characters "</"
   2. the appropriate case from the following:
      a. if *wildcarded(info)* is false, then if info[normalized prefix] is not no value, then the characters thereof, followed by a colon
      b. if *wildcarded(info)* is true, then if info[prefix] is not no value, then the characters thereof, followed by a colon
   3. the characters of info[local name]
   4. the character ">"

3. If *info* is an **attribute information item**, then *serialize(info)* is the in-order concatenation of the following:
   a. if info[omitted] is false, then
      i. the character space
      ii. the appropriate case from the following:
         1. if *wildcarded(info)* is false, then if info[normalized prefix] is not no value, then the characters thereof, followed by a colon

            2. if *wildcarded(info)* is true, then if info[prefix] is not no value, then the characters thereof, followed by a colon
        iii. the characters of info[local name]
        iv. the character "="
        v. the character quote
        vi. the appropriate case of the following:
            1. if the property info[prefix & schema normalized value] is present, then escape(info[prefix & schema normalized value])
            2. if info[schema normalized value] exists, then escape(info[schema normalized value])
            3. otherwise (the attribute was wildcarded), escape(info[normalized value])
        vii. the character quote
    b. otherwise, the empty octet sequence

4. If info is a **namespace information item**, then *serialize(info)* is the in-order concatenation of the following:
    a. if info[omitted] is false, then
        i. the character space
        ii. the characters "xmlns:"
        iii. the characters of info[prefix]
        iv. the character "="
        v. the character quote
        vi. escape(info[namespace name])
        vii. the character quote
    b. otherwise, the empty octet sequence

5. If *info* is an **unparsed entity information** item, then *serialize(info)* is the in-order concatenation of the following:
    a. if info[omitted] is false, then
        i. the characters "<!ENTITY"
        ii. the character space
        iii. info[name]
        iv. the character space
        v. the appropriate case of the following
            1. if info[public identifier] is not no value, then the in-order concatenation of the following:
                a. "PUBLIC"
                b. the character space
                c. info[public identifier]
                d. the character space

            e. info[system identifier]
        2. otherwise, the in order concatenation of the following:
            a. "SYSTEM"
            b. the character space
            c. info[system identifier]
    vi. if info[notation name] is not no value, then the in-order concatenation of the following:
        1. the character space
        2. "NDATA"
        3. the character space
        4. info[notation name]
    vii. the character ">"
b. otherwise, the empty octet sequence

6. If *info* is a **notation information item**, then *serialize(info)* is the in-order concatenation of the following:
    a. if info[omitted] is false, then
        i. the characters "<!NOTATION"
        ii. the character space
        iii. info[name]
        iv. the character space
        v. the appropriate case of the following
            1. if info[public identifier] and info[system identifier] are not both no value, then the in-order concatenation of the following:
                a. "PUBLIC"
                b. the character space
                c. info[public identifier]
                d. the character space
                e. info[system identifier]
            2. else if info[public identifier] has no value, the in-order concatenation of the following:
                a. "SYSTEM"
                b. the character space
                c. info[system identifier]
            3. otherwise, the in-order concatenation of the following
                a. "PUBLIC"
                b. the character space
                c. info[public identifier]
        vi. the character ">"
    b. otherwise, the empty octet sequence

7. Otherwise (this includes processing instruction, unexpanded entity reference, character, comment, and document type declaration information items, though characters and DTD's are accounted for by other means), *serialize(info)* is the empty sequence of octets.

3.5.2 The function recurse

The function *recurse* is a function which takes as input an ordered list *infos* of information items and proceeds as follows.

First, character information items in *infos* whose [omitted] property is 'true' are pruned by removing them from the list. Next, the pruned list is divided into an ordered sequence of sub-lists l_1 through l_k according to the rule that a sub-list which contains character items may not contain other types of information items, but otherwise *k* is as small as possible. The result of *recurse* is then the in-order concatenation of processing in order each sub-list l_i in turn in the following manner:

1. If l_i contains character information items, then let s_i be the string of characters of length equal to the size of l_i where the ISO 10646 character code of the *n*th character of s_i is equal to the [character code] property of the *n*th character of l_i. The output of processing l_i is then the result of the function invocation *escape*(s_i).
2. If l_i does not contain character information items, then the output of processing l_i is the in-order concatenation of *serialize(info)* as *info* ranges in order over the information items in the sub-list l_i .

3.5.3 The function escape

The function *escape* is that function which takes as input a string *s* and returns a copy of *s* where each occurrence of any of the five characters & < > ' " in *s* is replaced by its corresponding predefined entity.

3.5.4 The functions sort and compare

The function *sort* takes as input an unordered set or an ordered list of information items and returns an ordered list of those information items arranged in increasing order according to the function *compare*, unless some of the information items do not have a relative ordering, in which case a fatal error occurs.

The function *compare* takes two information items *a* and *b* as input and returns an element of {*less than or equal, greater than or equal, no relative ordering*} as output according to the following:

1. If *a* and *b* are both **attribute information items**, then (as in Canonical XML) *less than or equal* or *greater than or equal* is returned according to a lexicographical comparison with the [namespace name] property as the primary key and the [local name] as the secondary key.
2. If *a* and *b* are both **element information items**, then *less than or equal* or *greater than or equal* is returned according to a lexicographical comparison with the [namespace name] property as the primary key and the [local name] as the secondary key.
3. If *a* and *b* are both **namespace information items**, then *less than or equal* or *greater than or equal* is returned according to a lexicographical comparison with the [namespace name] property as the primary key and the [prefix] property as the secondary key.
4. If *a* and *b* are both **notation information items**, then *less than or equal* or *greater than or equal* is returned according to a comparison of their [name] properties
5. If *a* and *b* are both **unparsed entity information items**, then *less than or equal* or *greater than or equal* is returned according to a comparison of their [name] properties
6. Otherwise, *no relative ordering* is returned.

3.5.5 The function wildcarded

The function *wildcard* takes an element or an attribute information as input and returns a boolean indicating whether validation was not attempted on that item. In the Schema Centric Canonicalization algorithm, validation of an information item will only not be attempted as a consequence of the item or a parent thereof being validated against a wildcard whose {process contents} property is either *skip* or *lax*.

Let *i* be the information item input to *wildcarded*. The function is then defined as follows:

1. If *i*[validation attempted] is *none*, then *true* is returned.
2. Otherwise, *false* is returned.

3.5.6 The function wildcardOutputRoot

The function *wildcardOutputRoot* takes an element item as input and returns a boolean indicating whether the item is an appropriate one on which to place the contextual namespace declarations necessary for dealing with wildcarded items contained therein. Let *e* be the information item input to *wildcardOutputRoot.* The function is then defined as follows:

1. If *e*[omitted] is true, then *false* is returned.
2. If *wildcarded(e)* is *false* and *e*[attributes] contains any attribute items *a* for which *wildcarded(a)* is *true*, then *true* is returned.
3. If *wildcarded(e)* is *true*, and there does not transitively exist any [parent] element item *p* of *e* where either the preceding clause (2) applies or both *p*[omitted] is *false* and *wildcarded(p*) is *true,* then *true* is returned.
4. Otherwise, *false* is returned.

4. Use of Schema Centric Canonicalization in XML Security

4.1 Algorithm Identification

The XML-Signature Syntax and Processing recommendation (XML DSIG) defines the notion of a canonicalization algorithm together with the use of URIs as identifiers for such algorithms. In XML DSIG, the use of canonicalization algorithms is architected in three places:

1. As part of the signature generation and validation processes, where it is used to canonicalize a SignedInfo element prior to its being fed into a digest algorithm.
2. As a Transform algorithm in the pipeline of Transforms inside a Reference, used to modify data during the reference generation and validation processes. As a matter of good XML DSIG hygiene, such a canonicalization Transform should always be used in the pipeline, and in fact should always occur as the last Transform therein.
3. As the means by which a Transform in the pipeline which requires an octet stream as input but is instead presented (by the previous Transform) with an input node-set converts the latter into the former.

XML Encryption makes similar use of these algorithms.

This specification asserts that the URI of the Schema Centric Canonicalization algorithm namespace is the identifier (in the sense of XML DSIG) of a canonicalization algorithm. This identifier denotes the Schema Centric Canonicalization algorithm. The algorithm does not require or permit any explicit parameters.

4.2 Re-Enveloping of Canonicalized Data

As is discussed in Exclusive XML Canonicalization, many applications from time to time find it useful to be able to change the enveloping context of a subset of an XML document without changing the canonical form thereof.

In such situations, if Schema Centric Canonicalization is the algorithm of relevance, then applications SHOULD avoid references to notations or unparsed entities in the document subset in question, since the canonical representation of the notation and entity declarations referred to (which must, for security, be part of the canonical form) are defined in a document type declaration, the presence of which significantly complicates the task of re-enveloping.

5. Resolutions

This section discusses a few key decision points as well as a rationale for each decision.

5.1 No Non-Schema-Influencing Information Items

Several of the eleven different types of information items either can never appear in an infoset which successfully validates according to XML Schema or can in no way affect the outcome thereof. Accordingly, representations of such information items never appear in the output of the Schema Centric Canonicalization algorithm. These types of information item are the following:

1. **comment** information items and **processing instruction** information items: as is described in the XML Schema Structures recommendation, comments and processing instructions, even in the midst of text, are ignored for all validation purposes. Thus, for example, each can appear in such places as the middle of the sequence of digits of an integer which is the content of an element with an integral simple type. Were it required (or even optional) to preserve the significance of such items with respect to the canonicalization, applications, particularly those wishing to shred XML information into a relational or other store, would face cumbersome and significant impediments to implementation.
2. **unexpanded entity reference** information items: as is explained in the XML Infoset recommendation, a validating XML processor will never generate unexpanded entity reference information items for a valid document.
3. **document type declaration** information items: these are excluded since all possible effects of their processing are modeled in various properties of other information items.

5.2 No Special Whitespace Processing

Believing their reasoning to be sound, we adopt the attitude of Canonical XML towards the processing of whitespace in character content, namely that no special processing is carried out:

> "All whitespace within the root document element MUST be preserved (except for any #xD characters deleted by line delimiter normalization). This includes all whitespace in external entities."

Moreover, for analogous reasons, we adopt the attitude of Exclusive XML Canonicalization towards the lack of special processing of the xml:lang and the xml:space attributes.

It is perhaps worth noting by way of contrast that (unrelated to xml:lang and xml:space) XML Schema defines certain whitespace processing rules of its own; these are, of course, carried out by Schema Centric Canonicalization.

5.3 Case-Mapping vs. Case-Folding

The Unicode Technical Report on Case Mappings distinguishes case-mapping from a similar process termed case-folding. Unlike case-mapping, case-folding is a locale-independent operation, and does not encounter the issue that strings may be equal or differ depending on the direction in which they are case-mapped. As is clear in the report, case-folding suffers from being only an approximation to language-specific rules of processing, and is primarily aimed at legacy systems where locale information simply is not feasibly available with which to do a more complete processing.

The Schema Centric Canonicalization algorithm supports the use of either case-mapping or case-folding in user schemas.

5.4 No Canonicalization of anyURI Datatype

XML Schema Datatypes does not define a canonical lexical representation for data of type anyURI. In the present specification, thought was given to reconsidering this position. As is described in the specification of Uniform Resource Identifiers, various aspects of the syntactic structure of URIs are considered case insensitive: the scheme part of the URI is an example (or *probably* is one: contrast §3.1 with §6 in RFC2396 with respect to this point), and various particular schemes have substructure that is so. Moreover, a subset of URI share a common syntax for representing hierarchical relationships within their namespace, and for the relative (as opposed to absolute) form of such URI, an algorithm exists (see §5.2 of RFC2396) by which certain aspects of the URI representation involving "." and ".." are canonicalized.

For these and related reasons it is reasonable to ask whether a canonical lexical representation for data of type anyURI should be specified. This, however, is a difficult if not insurmountable task. Many of the details of an appropriate canonicalization (such as case-mapping or case-folding) are inherently scheme-specific, and it is intrinsically impossible for any one Schema Centric Canonicalization implementation to understand the universe of possible URI schemes it might encounter (and so canonicalize them all appropriately). Even for some commonly known URI schemes, the relevant specifications lack crisp clarity on some germane issues. And the algorithm of §5.2 of RFC2396 can (see *ibid*, §5.1) only be carried out in the context of a specific base URI; as generally speaking such relevant base URI may be application-level notions not represented in XML, the algorithm of §5.2 must remain out of scope so far as XML canonicalization is concerned.

These reasons, together with the lack of compelling pragmatic problems caused by simply having all anyURI data canonicalize to itself, indicate that the prudent course of action is that Schema Centric Canonicalization should not differ from XML Schema Datatypes on this issue.

6. References

Keywords

RFC 2119. *Key words for use in RFCs to Indicate Requirement Levels*. Best Current Practice. S. Bradner. March 1997. S. Bradner. March 1997. http://www.ietf.org/rfc/rfc2119.txt

Unicode

Unicode 3.1. The Unicode Consortium. http://www.unicode.org/unicode/reports/tr27/.

Unicode Normalization

Unicode Normalization Forms. The Unicode Consortium. http://www.unicode.org/unicode/reports/tr15/.

Unicode Case Mappings

Case Mappings. The Unicode Consortium. http://www.unicode.org/unicode/reports/tr21/.

URI

RFC 2396. *Uniform Resource Identifiers (URI): Generic Syntax.* T. Berners-Lee, R. Fielding, L. Masinter. August

1998. See also RFC 2732. *Format for Literal IPv6 Addresses in URL's*. R. Hinden et al. http://www.ietf.org/rfc/rfc2396.txt. See also http://www.ietf.org/rfc/rfc2732.txt.

XML
Extensible Markup Language (XML) 1.0 (Second Edition). W3C Recommendation. T. Bray, E. Maler, J. Paoli, C. M. Sperberg-McQueen. October 2000. http://www.w3.org/TR/2000/REC-xml-20001006.

XML-C14N
Canonical XML. W3C Recommendation. J. Boyer. March 2001. http://www.w3.org/TR/2001/REC-xml-c14n-20010315 http://www.ietf.org/rfc/rfc3076.txt

XML-DSig
XML-Signature Syntax and Processing. W3C Recommendation. D. Eastlake, J. Reagle, and D. Solo. 12 February 2002. http://www.w3.org/TR/2002/REC-xmldsig-core-20020212/

XML-Enc
XML Encryption Syntax and Processing. D. Eastlake, and J. Reagle. W3C Candidate Recommendation. 04 March 2002. http://www.w3.org/TR/2002/CR-xmlenc-core-20020304

XML-Exc-C14N
Exclusive XML Canonicalization W3C Candidate Recommendation. J. Boyer, D. Eastlake, and J. Reagle. 12 February 2002. http://www.w3.org/TR/2002/CR-xml-exc-c14n-20020212

XML-Infoset
XML Information Set, John Cowan and Richard Tobin, eds., W3C, 24 October 2001. See http://www.w3.org/TR/2001/REC-xml-infoset-20011024/

XML-NS
Namespaces in XML. Recommendation. T. Bray, D. Hollander, and A. Layman. January 1999. http://www.w3.org/TR/1999/REC-xml-names-19990114/

XML-Schema
XML Schema. Recommendation. H. Thompson, D. Beech, M. Maloney, N. Mendelsohn. 2 May 2001. http://www.w3.org/XML/Schema

XML-Schema-Errata
XML Schema 1.0 Specification Errata. http://www.w3.org/2001/05/xmlschema-errata

XPath

XML Path Language (XPath) Version 1.0 , W3C Recommendation. eds. James Clark and Steven DeRose. 16 November 1999. http://www.w3.org/TR/1999/REC-xpath-19991116.

7. Revision History

13 February 2002	Initial distribution for public review.
15 May 2002	Update references to XML-DSIG, XML-Exc-C14N, XML-Enc to refer to updated publications. Revise commentary thereon in introduction to reflect changes in these updates. Allow embeddedLang attribute to be used on schema instances as well as schemas themselves. Added by-fiat identification of uses of XPath in XML Schema itself. Clarified position with respect to (non)canonicalization of anyURI. Expanded limitations section per observations of Exclusive XML Canonicalization and others. QNames (and derivations and lists thereof) were not being namespace prefix desensitized. Fixed. Added pointer to supporting .xsd file.
20 May 2003	Reformated per OASIS requirements. A very few minor editorial fixes, updated legal language.

Appendix A: Notices